Computer Languages, Systems & Structures 44 (2015) 218–237

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

On the verification of architectural reconfigurations

Alejandro Sanchez [a,b,*], Alexandre Madeira [b], Luís S. Barbosa [b]

[a] Universidad Nacional de San Luis, Ejército de los Andes 950, D5700HHW Argentina
[b] HASLab INESC TEC & Universidade do Minho, 4710-057 Portugal

ARTICLE INFO

Article history:
Received 6 February 2015
Received in revised form
30 May 2015
Accepted 10 July 2015
Available online 17 July 2015

Keywords:
Architectural reconfiguration
Architectural description language
Modal logic
Graded hybrid logic

ABSTRACT

In a reconfigurable system, the response to contextual or internal change may trigger reconfiguration events which, on their turn, activate scripts that change the system's architecture at runtime. To be safe, however, such reconfigurations are expected to obey the fundamental principles originally specified by its architect. This paper introduces an approach to ensure that such principles are observed along reconfigurations by verifying them against concrete specifications in a suitable logic. Architectures, reconfiguration scripts, and principles are specified in Archery, an architectural description language with formal semantics. Principles are encoded as constraints, which become formulas of a two-layer graded hybrid logic, where the upper layer restricts reconfigurations, and the lower layer constrains the resulting configurations. Constraints are verified by translating them into logic formulas, which are interpreted over models derived from Archery specifications of architectures and reconfigurations. Suitable notions of bisimulation and refinement, to which the architect may resort to compare configurations, are given, and their relationship with modal validity is discussed.

1. Introduction

The purpose of dynamic architectural reconfiguration [1] is to maintain the quality level of a system as contextual or internal conditions vary. This is primarily achieved through a combination of sensors, which somehow measure the system, and actuators, *i.e.*, scripts which modify the system's architecture under specified situations. Reconfigurations, however, may disrupt the basic design principles, originally fixed by the architectural patterns in use. Therefore, a mechanism is required to ensure that emerging reconfigurations conform to the design principles, regardless of how they take place.

This paper introduces an approach to provide such a mechanism. It focuses on reconfigurations that constitute undesired sequences of change, or that lead to forbidden configurations. Design principles are specified as formulas in a modal logic, and then are verified against models of reconfigurations. For this, we extend an *architectural description language* (ADL) [2] called Archery [3,4], which is a *domain specific language* [5] used to animate, analyse and verify system's architectures. It is organized as a core and a number of modules. The core is for modelling architectures in terms of architectural patterns, and the modules are for specifying constraints and reconfiguration scripts. A constraint restricts either structure, behaviour or possible reconfigurations of a system. Reconfiguration scripts are executed by a configuration manager when conditions, specified as constraints, hold. The language semantics is given by a translation into a process algebra [3], for the behavioural part, and by an encoding into bigraphical reactive systems [4], for the structural part. Constraints are translated into a modal

* Corresponding author.
E-mail addresses: asanchez@unsl.edu.ar (A. Sanchez), madeira@di.uminho.pt (A. Madeira), lsb@di.uminho.pt (L.S. Barbosa).

http://dx.doi.org/10.1016/j.cl.2015.07.001
1477-8424/

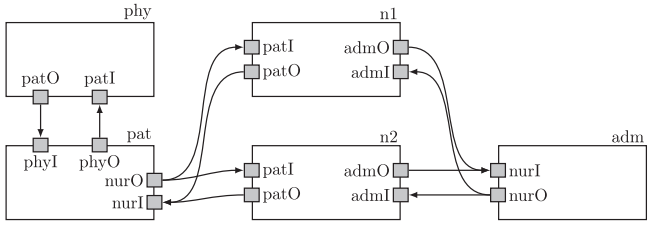

Fig. 1. A configuration with two nurses.

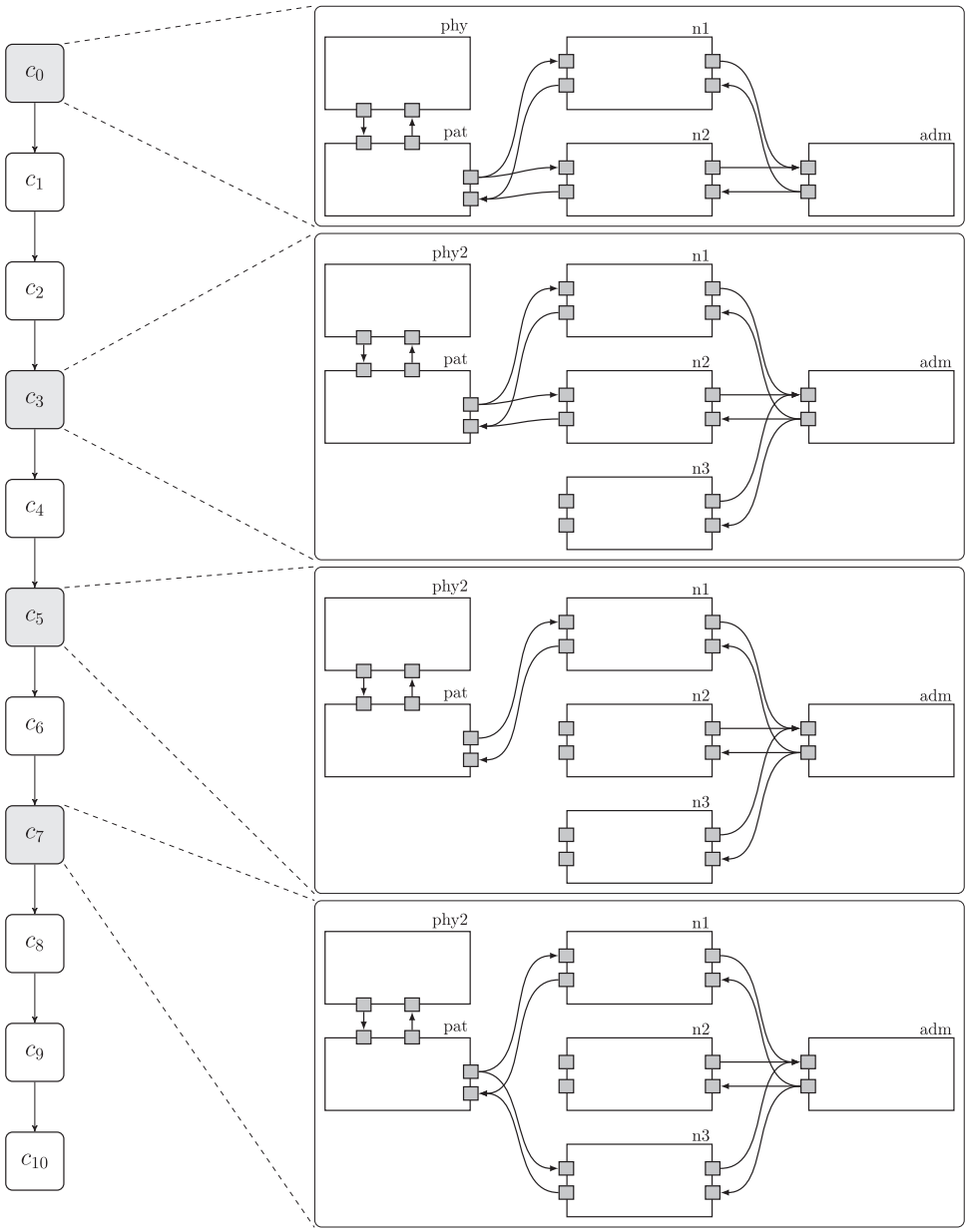

Fig. 2. A nurse being substituted.

logic and are verified against models derived from architectural specifications [6–8]. We extend Archery's syntax and the underlying mathematical framework to support models and properties of reconfigurations.

The underlying logic proposed here is a *two-layer graded hybrid logic*. As usual in modal logic, models are relational structures over a state space (whose elements are called *worlds*, states, or points). Being *hybrid*, the logic is equipped with

both nominals and a reference operator. The former is a proposition that is only satisfied at the world it identifies. The latter constrains a formula to hold at the world named by a specified nominal. Together they make possible to express, for instance, that two worlds are identical, or that their relationship is irreflexive. Other features of the logic are its *hierarchical* character [9], which enforce two layers of description, the use of graded modalities to describe the cardinality of relations, and operations to select and iterate over a set of relations. Hierarchical formulas allow us to describe models organized into layers of abstraction, which result from the common practice of refining a world into a more elaborated model. Data parameters are also allowed in relations.

Notions of simulation and bisimulation are introduced for models of this logic. They provide refinement and equivalence relations, respectively, in order to discuss whether a script can replace, or be interchanged, with another.

Global properties of the logic are studied. Reconfiguration scripts are said to be equivalent if, and only if, they satisfy the same formulas. Preservation of modal equivalence by bisimilarity is proved, and a full Hennessy–Milner like theorem arises for the non-graded fragment.

In this context, the contributions of the article are an ADL to define two-level constraints whose first level describes reconfigurations and the second, the resulting configurations; the characterization of the associated logic as well as of suitable notions of model bisimulation and refinement; a derivation of models from architectural specifications; and finally a translation that takes constraints and yields a formula in the two-layer graded hybrid logic. Behavioural constraints and triggers are not dealt here; the interested reader is referred to the first author forthcoming thesis [10].

The approach proposed here is illustrated with a fragment of the *blood transfusion process*, the architecture of a medical procedure which is critical in the sense that it may involve risk to patients [11]. In particular, it requires ensuring blood compatibility between the donor and the patient, since an incompatible transfusion can cause a reaction with fatal consequences. A major source of these incidents is misidentification, which might occur at stages of the process that require checking patient's identity, or handling material with patient identification data. Architectural principles are laid down to prevent misidentification, and the approach is used to verify them.

Organization: After describing the blood transfusion example in Section 2, a background summary of the Archery language is provided in Section 3, Section 4 presents the logic, introduces bisimulation and refinement relations, and studies their properties. Section 5 introduces the derivation of models, the translation of constraints, and illustrates the approach with the verification of some constraints from the example. Then Section 6 describes related work, and Section 7 sums up and mentions ongoing and future work.

2. A blood transfusion process: avoiding misidentification

The example process starts after a blood transfusion is prescribed to a patient and ends when the patient is discharged. The procedure is supported by a software system, accessed through mobile devices by the involved staff that includes a physician, nurses, and the administration. It requires collecting a blood sample from the patient, establishing blood group and factor, selecting suitable blood units, performing the actual transfusion, and monitoring the patient for a given period of time. The patient is discharged when such period ends without any adverse reaction being observed. Fig. 1 depicts an example configuration using an informal notation, in which white rectangles represent components, small grey rectangles, ports, and arrows, interactions. It includes a patient (`pat`), a physician (`phy`), two nurses (`n1` and `n2`), and the administration (`adm`).

The staff, however, may change during the procedure. Nurses can enter (leave, resp.) the ward, and can be assigned (unassigned, resp.) to (from, resp.) a blood transfusion patient. These changes must avoid configurations in which a misidentification is more likely to occur, and proceed under the supervision of the administration.

Consider, for instance, the sequence of reconfigurations represented by the transition system on the left of Fig. 2 that substitutes nurse `n2` with `n3`. We focus on configurations c_0, c_3, c_5, and c_7, which are refined into (informal) architectural diagrams on the right. Configuration c_0 is similar to that described in Fig. 1. In configuration c_3, a nurse (`n3`) has entered the ward and checked in with administration, the latter represented by the two connections between `n3` and `adm`. Then, nurse `n2` is unassigned from patient `pat`, which is represented in c_5 by the absence of connections between them. Subsequently, nurse `n3` is assigned to the patient and the result is shown in configuration c_7. The sequence finalizes with nurse `n2` leaving the ward (removed from the configuration).

Several *architectural principles* are defined to prevent configurations in which a misidentification might occur. It is requested that a physician and at least two nurses are assigned to each patient undergoing a blood transfusion. The physician orders and monitors the procedure, one nurse leads it, and the other assists the former to prevent a misidentification. Then, it must be kept invariant in every configuration that any patient must have (S1) a designated physician; and (S2) at least two nurses assigned. Changes, on the other hand, must observe the following constraints: (R1) upon entering a ward, a nurse must check-in with administration before receiving any patient assignment; (R2) a nurse cannot be unassigned from a patient if less than two nurses are left assigned; and (R3) a nurse must have no assignments before checking out with administration and leaving.

In the example, configuration c_5 does not satisfy S2. We use reconfiguration constraints to ensure that changes observe these requirements, and avoid error-prone situations.

$$
\begin{array}{lll}
\mathit{Pat} & ::= & \textbf{pattern}\ \mathrm{TYPE}\ \mathrm{Elem}+\ \mathit{Const}^*\ \textbf{end} \\
\mathit{Elem} & ::= & \textbf{element}\ \mathrm{TYPE}\ \mathit{ElemInt}\ \mathit{Beh}? \\
\mathit{ElemInt} & ::= & \textbf{interface}\ \mathit{Port}+ \\
\mathit{Port} & ::= & (\textbf{in}|\textbf{out})\ (\textbf{and}|\textbf{or}|\textbf{xor})\ \mathit{IdList}\ (:\mathit{Domain})?\textbf{;}
\end{array}
$$

Fig. 3. Syntax for patterns.

$$
\begin{array}{lll}
\mathit{Var} & ::= & \mathrm{VAR}\ \textbf{:}\ \mathrm{TYPE} = \mathit{Inst}\ \textbf{;} \\
\mathit{Inst} & ::= & (\ \mathit{ElemInst} \mid \mathit{PatInst}\ ) \\
\mathit{ElemInst} & ::= & \mathrm{TYPE}\ (\ \mathit{DataExprs}?\ ) \\
\mathit{PatInst} & ::= & \textbf{architecture}\ \mathrm{TYPE}\ \mathit{Body}\ \mathit{Const}^*\ \textbf{end} \\
\mathit{Body} & ::= & \mathit{Insts}?\ \mathit{Atts}?\ \mathit{ArchInt}? \\
\mathit{Insts} & ::= & \textbf{instances}\ \mathit{Var}+ \\
\mathit{Atts} & ::= & \textbf{attachments}\ \mathit{Att}+ \\
\mathit{Att} & ::= & \textbf{from}\ \mathit{PortRef}\ \textbf{to}\ \mathit{PortRef}\ \textbf{;} \\
\mathit{ArchInt} & ::= & \textbf{interface}\ \mathit{Ren}+ \\
\mathit{Ren} & ::= & \mathit{PortRef}\ \textbf{as}\ \mathrm{ID}\ \textbf{;} \\
\mathit{PortRef} & ::= & \mathrm{VAR.ID}
\end{array}
$$

Fig. 4. Syntax for instances.

3. The Archery language

3.1. Architectures

The specification of an architecture comprises one or more (architectural) patterns, a main architecture, and data specifications. A *pattern* defines (architectural) elements (components and connectors) and might have associated constraints (see syntax in Fig. 3). For instance, pattern `Ward` defines elements `Patient`, `Physician`, `Nurse`, and `Administration` in Listing 1 to represent configurations that carry out blood transfusions.

Listing 1. Blood transfusion pattern.

```
1  pattern Ward()
2  element Patient()
3    interface in xor phyI, nurI; out xor phyO, nurO;
4  element Physician()
5    interface in xor patI; out xor patO;
6  element Nurse()
7    interface in xor patI, admI; out xor patO, admO;
8  element Administration()
9    interface in xor nurI, phyI; out xor nurO, phyO;
10 end
```

Each *element* includes an *interface* that contains one or more *ports*, each of which is defined by a polarity, a port type, and a name. The polarity indicates how communication among attached ports flows, and can be either `in` or `out`. Ports are synchronous: actually a suitable process algebra expression can be used to emulate any other port behaviour. The port type indicates how many participants are necessary for a communication to take place, and can be either `and`, `xor`, or `or`. While an `and` port requires all attached participants to synchronise, a `xor` port requires exactly one. In between, an interaction with an `or` port requires at least one, but it may include any number of participants. For instance, the interface of `Administration` defines `xor` ports `patI` and `patO`. An element can optionally include a behaviour: a set of actions, and a set of process descriptions expressed in a subset of the mCRL2 process algebra. The sequel focuses on the structural dimension and excludes such behavioural specifications.

Table 1
Reconfiguration operations.

Name	Syntax	Description
Create variable	`v:T`	Creates variable `v` of type `T`
Destroy variable	`destroy(v)`	Destroys variable `v`
Create instance	`v=T()`	Creates an instance of type `T` and leaves it in `v`
Destroy instance	`clear(v)`	Destroys any instance in variable `v`
Attach	`attach(f,o,t,i)`	Attaches port `o` of instance in `f` to port `i` of instance in `t`
Detach	`detach(f,o,t,i)`	Removes attachment that goes from port `o` of instance in variable `f` to port `i` of instance in variable `t`
Add renaming	`show(v,p,q)`	Renames port `p` in variable `v` to `q`
Remove renaming	`hide(v,q)`	Removes renaming `q` of architecture in variable `v`
Move instance	`imove(s,t)`	Whatever is referred by variable `s` becomes referred by `t`; the reference to the contents of `t` is lost, but its attachments and renamings remain
Move variable	`vmove(v,a)`	Moves variable `v` to the architecture in variable `a`

Instances – architectures and element instances – are defined according to the syntax in Fig. 4. They are stored in variables that are defined by an identifier and a type that must match an element or pattern name. See, for instance, line 1 of Listing 2. Allowed values are instances of a type (element or pattern), that do not necessarily need to match the variable's own type.

Listing 2. A configuration for performing a blood transfusion.

```
1  w:Ward=architecture Ward()
2  instances
3   pat:Patient=Patient(); phy:Physician=Physician();
4   n1:Nurse=Nurse(); n2:Nurse=Nurse();
5   adm:Administration=Administration();
6  attachments
7   from pat.phyO to phy.patI; from phy.patO to pat.phyI;
8   from n1.patO to pat.nurI; from pat.nurO to n1.patI;
9   from n2.patO to pat.nurI; from pat.nurO to n2.patI;
10  from n1.admO to adm.nurI; from adm.nurO to n1.admI;
11  from n2.admO to adm.nurI; from adm.nurO to n2.admI;
12 end
```

An architecture describes the configuration a set of instances adopt. It contains a token that must match a pattern name, a set of variables, an optional set of attachments, and an optional interface. The type of each variable in the set is limited to an element in the pattern the architecture is instance of. An attachment indicates which output port communicates with which input port; each includes port references to an output and to an input port. A port reference is an ordered pair of identifiers: the first one matching a variable identifier, and the second matching a port of the variable's instance. For instance, the attachments in the example configuration connect two nurses with a patient and the administration. The architecture interface is a set of one or more port renamings. Each port renaming contains a port reference and a token with the external name of the port. Ports not included in this set are not visible from the outside. An architecture can have associated constraints, which are defined as described in Section 3.3.

3.2. Reconfigurations

Reconfiguration scripts are sequences of operations that affect the structure of architectures. Configuration managers execute them when the associated triggering conditions are met. They also have the ability to stop, reconfigure, and restart architectures from a given state.

Reconfiguration operations are devoted to the creation and removal of instances, attachments, renamings and variables, as well as to the movement of instances. Table 1 shows name, format, and a brief description of them.

Scripts that change configurations by performing a blood transfusion are shown in Listing 3. Moving a `Nurse` (instance) into a `Ward` (configuration) represents allowing the nurse in. Attaching a `Nurse` to `Administration` materializes checking the nurse in. Similarly, if the attachment is to a `Patient`, it represents assigning the nurse to the patient. These scripts are triggered by constraints that describe behaviour the participants show.

$$
\begin{array}{lll}
Const & ::= & SConst \mid BConst \mid RConst \\
SConst & ::= & \textbf{structural constraint } FDecl+ \textbf{ end} \\
RConst & ::= & \textbf{reconfiguration constraint } FDecl+ \textbf{ end}
\end{array}
$$

Fig. 5. Constraint types.

Listing 3. Reconfiguration scripts for nurse management.

```
1  script enter(w:Ward; n:Nurse) vmove(n,w); end
2  script leave(n:Nurse) vmove(n,root); end
3  script checkIn(n:Nurse;a:Administration)
4   attach(n,admO,a,nurI); attach(a,nurO,n,admI);
5  end
6  script checkOut(n:Nurse;a:Admininistration)
7   detach(n,admO,a,nurI); detach(a,nurO,n,admI);
8  end
9  script assign(p:Patient, n:Nurse)
10  attach(n,patO,pat,nurI); attach(pat,nurO,n,patI);
11 end
12 script unassign(p:Patient, n:Nurse)
13  detach(n,patO,pat,nurI); detach(pat,nurO,n,patI);
14 end
```

For instance, the sequence of reconfigurations shown in Fig. 2 is the result of executing such scripts as follows:

- `enter(w,n3)` lets nurse `n3` in,
- `checkIn(n3,adm)` checks `n3` in (conf. c_3),
- `unassign(pat,n2)` unassigns nurse `n2` from the patient (conf. c_5),
- `assign(pat,n2)` assigns `n2` to the patient (conf. c_5), and
- `leave(n2)` lets nurse `n2` out of the ward.

3.3. Constraints

The constraint language allows us to precisely describe design decisions by associating constraints to a pattern, or to a pattern instance [7]. A *constraint* restricts design dimensions – structure, behaviour, or reconfigurations – through one or more formula declarations (see Fig. 5).

The language for declaring formulas is defined generically. The actual languages are instances of it, obtained by making constructs specific to the design dimension that is intended to be restricted. We present the generic language and instances for specifying both (i) constraints over structure; and (ii) two-layer constraints whose upper and lower layers restrict reconfigurations and obtained structures.

3.3.1. Two-layer generic formulas

A formula is interpreted from a local and internal point of view over a model $\mathfrak{M}$ – a labelled graph whose nodes W are called worlds. Each edge is labelled by a *modal symbol* M, taken from a set *Mod*. A *modal symbol* identifies an *accessibility relation* $R[M]$ in $W \times W$, where an ordered pair (w_1, w_2) indicates that it is possible to access w_2 from w_1 through an edge with label M. Formulas are interpreted at a specific world within $\mathfrak{M}$. But note that in a two-layer model $\mathfrak{M}$, there is a model $\mathfrak{M}'$ associated to each w in $\mathfrak{M}$.

Actual models are derived from specifications of architectures and their reconfigurations. The resulting graphs correspond to a metamodel given by the dimension being restricted [12]. The two corresponding metamodels are presented as a class diagram, and define the structure of such graphs. The diagrams present the types of nodes and how their instances are related. Modal symbols (relation labels in the diagram) name relations between either constituents of an architecture, or two configurations, where the second is obtained upon executing an operation to the first, in a reconfiguration sequence. In the latter case, a syntax for the structure of symbols, which represent the execution of reconfiguration operations, is also provided.

The language provides *symbol terms* `SymT` to match symbols according to their structure. They are built upon atomic symbols and filters. A filter is either a path identifying a configuration variable, a configuration variable type, a variable for

$$
\begin{array}{lll}
\mathit{SymT} & ::= & \text{ASYM } (\text{[} \mathit{Filter}\ (,\mathit{Filter})^*\ \text{]})? \\
\mathit{Filter} & ::= & \mathit{Arg} \mid \text{ID} \\
\mathit{Arg} & ::= & \mathit{Path} \mid \text{CVAR} \mid \text{TYPE} \\
\mathit{Path} & ::= & \text{VAR } (.\mathit{Path})? \\
\mathit{SymF} & ::= & \mathit{SymT} \mid \textbf{true} \mid \textbf{false} \mid \textbf{not } \mathit{SymF} \\
 & \mid & \mathit{SymF} \textbf{ and } \mathit{SymF} \mid \mathit{SymF} \textbf{ or } \mathit{SymF} \\
 & \mid & \textbf{exists } \text{CVAR}(:\text{TYPE})?.\mathit{SymF} \mid \textbf{forall } \text{CVAR}(:\text{TYPE})?.\mathit{SymF}
\end{array}
$$

Fig. 6. Symbol formulas.

$$
\begin{array}{lll}
\mathit{FDecl} & ::= & \text{ID } \mathit{CPars}? = F; \\
\mathit{CPars} & ::= & (\ \text{CVAR:TYPE } (,\ \text{CVAR:TYPE })^*\ ) \\
F & ::= & \text{P} \mid \textbf{true} \mid \textbf{false} \\
 & \mid & \textbf{not } F \mid F \textbf{ or } F \mid F \textbf{ and } F \mid F \textbf{ implies } F \mid F \textbf{ iff } F \\
 & \mid & [\mathit{RelF}]\ F \mid <\mathit{RelF}> F \mid [\text{N},\mathit{RelF}]\ F \mid <\text{N},\mathit{RelF}>\ F \\
 & \mid & \text{I} \mid \textbf{at } \text{I } F \mid \text{CVAR} \mid \textbf{at } \text{CVAR } F \\
 & \mid & \textbf{exists } \text{CVAR}(:\text{TYPE})?.F \mid \textbf{forall } \text{CVAR}(:\text{TYPE})?.F \\
 & \mid & \textbf{nest } \text{ID } \mathit{ACPars}? \\
\mathit{ACPars} & ::= & (\ (\mathit{Path} \mid \text{CVAR})\ (,\ (\mathit{Path} \mid \text{CVAR}))^*\ ) \\
\mathit{RelF} & ::= & \mathit{SymF} \mid \mathit{RelF}.\mathit{RelF} \mid \mathit{RelF}+\mathit{RelF} \mid \mathit{RelF}^* \mid \mathit{RelF}+
\end{array}
$$

Fig. 7. Formula declarations.

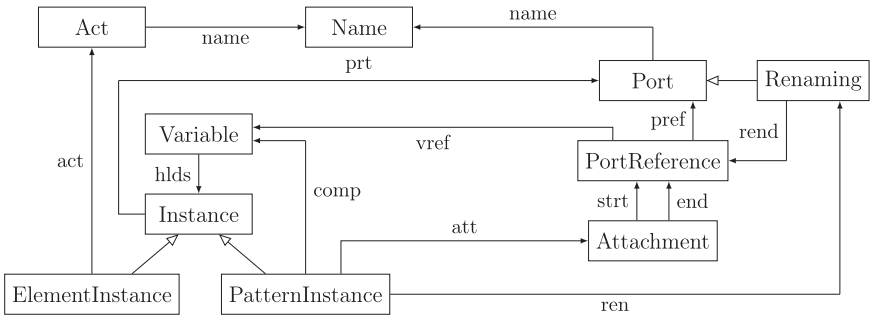

Fig. 8. Metamodel for architectures: structure.

configuration variables, or a port identifier. Actual atomic symbols and filter combinations depend on the language instantiation.

Symbol formulas `SymF` allow for selecting and operating upon sets of symbols, and binding variables in symbol terms (see Fig. 6). A symbol formula can be either a symbol term, a constant, a negated symbol formula, a conjunction, a disjunction, or a quantifier. Constants **`true`** and **`false`** represent the universe (*Mod*) and the empty sets, respectively. Negation, conjunction and disjunction represent the corresponding set operations of complement, intersection and union. Quantifiers bind configuration variables in filters. A type in a filter is a simplified version of an existential quantifier. The relations that a `SymF` selects are called `SymF`−relations.

Formula declarations consist of an identifier, optional configuration parameters, and a formula (see Fig. 7). Configuration parameters are used to let pass identifiers of configuration variables to a formula.

A *formula* `F` is either a propositional formula, a modal formula, a graded modality formula, a hybrid formula, a quantifier, or a nested formula. Propositional, modal, and graded formulas describe the model in terms of the underlying metamodel. A proposition characterizes a feature in a given world. Modal and graded operators scan worlds according to accessibility relations passed as a parameter in the form of a relation formula `RelF`.

Relation formulas combine relations. A relation formula `RelF` is either a symbol formula, a concatenation, a union, or an iteration. A concatenation returns the composition of two relations, and a union their union. An iteration is the successive concatenation of relations defined by a relation formula. There are two types: one that admits the absence of the relation, and one that requires at least one occurrence of it.

In a *modal formula*, a *possibly formula* ⟨`RelF`⟩ `F` indicates that the present world is `RelF`−related with another world satisfying (formula) `F`, whereas a *necessarily formula* [`RelF`] `F` indicates that any `RelF`−relationship from the present world leads to a world satisfying `F`.

Similarly, *graded modality formulas* come in two flavours as well. An *at least formula* ⟨n, `RelF`⟩`F` that is satisfied at worlds where `F` holds in at least `n+1` `RelF`−related worlds, and an *all but formula* [n, `RelF`] `F` that describes worlds where `F` holds in all but at most `n` `RelF`−related worlds.

Fig. 9. Metamodel for reconfiguration scripts.

$$
\begin{array}{rcl}
SymT & ::= & \textbf{vnew}([Arg])? \;\; // \text{ create variable} \\
& | & \textbf{destroy}\ ([Arg])? \;\; // \text{ destroy variable} \\
& | & \textbf{inew}\ ([Arg?(,\ \text{TYPE})?])? \;\; // \text{ create instance} \\
& | & \textbf{clear}([Arg])? \;\; // \text{ destroy instance} \\
& | & \textbf{attach}([Arg?(,\ \text{ID})?(,\ Arg)?(,\ \text{ID})?])? \;\; // \text{ attach} \\
& | & \textbf{detach}([Arg?(,\ \text{ID})?(,\ Arg)?(,\ \text{ID})?])? \;\; // \text{ detach} \\
& | & \textbf{show}([Arg?(,\ \text{ID})?(,\ \text{ID})?])? \;\; // \text{ detach} \\
& | & \textbf{hide}([Arg?(,\ \text{ID})?])? \;\; // \text{ remove renaming} \\
& | & \textbf{imove}([Arg?(,\ Arg)?])? \;\; // \text{ move instance} \\
& | & \textbf{vmove}([Arg?(,\ Arg)?])? \;\; // \text{ move variable}
\end{array}
$$

Fig. 10. Syntax for reconfiguration constraints.

Hybrid formulas are built of a *nominal* `I`, which is satisfied if the current world is the unique world referenced by such `I`, and of a *reference operator* **at** `I F`, satisfied if at the world named by `I`, `F` is.

A *nested formula* consists of an identifier, and actual configuration parameters, and describes nested models. The usage of configuration parameters by nested constraints depends on the restricted dimension. Language instances vary in the constructs they offer, as it is indicated on their introduction.

3.3.2. Structural constraints

Models for interpreting structural constraints are derived from architectures, according to the metamodel shown in Fig. 8. Worlds are instances, ports, actions, variables, port references, attachments, names, and renamings. The relationships among worlds conform the relations in function *R*, and their labels become plain atomic modal symbols ASYM. For convenience, the extra symbols `attd` and `evt` are also included. The former identifies the relationship between two worlds representing variables connected through an attachment. It is obtained as $R[\texttt{vref}]^{\circ} \circ R[\texttt{strt}]^{\circ} \circ R[\texttt{end}] \circ R[\texttt{vref}]$, where $R[s]^{\circ}$ denotes the converse of a relation. The latter is obtained as $R[\texttt{prt}] \cup R[\texttt{act}]$.

Propositions are classified as follows: (*a*) *Naming propositions* that hold when evaluated at a (world) *w* representing an action or port with their name. They exist for each name used in actions and ports. (*b*) *Meta-type propositions* that hold when *w* belongs to a specific participant set, *e.g.*, `PatternInstance`. (*c*) *Emptiness proposition* (namely `Empty`) that holds when *w* is a variable with no associated instance. (*d*) *Type propositions* that test if *w* is an instance or a variable of a type in the specification. For example, the Ward pattern generates propositions `Ward`, `Physician`, `Patient`, `Nurse`, and `Administration`.

Each variable in an architectural specification defines a nominal in the set *Nom*. Each nominal holds exactly at the world that represents the corresponding variable. In addition, they are also included in a subset Nom_{TYPE}, depending on the variable's type. Variables and parameters in formulas are bound to nominals.

Structural constraints that specify requirements S1 and S2 are shown in Listing 4. In both cases, a parameter `p` receives a nominal referencing (a world that represents) a patient. The first constraint requires the specific patient to have an attached physician. It uses a reference operator that holds if the rest of the constraint holds at the patient. This happens when a physician is attached to the patient in the configuration, which is indicated with a possibly operator for the relation that attachments in configurations give rise to. The second constraint requires the patient to have at least two nurses assigned. The *at least* operator is used in this case to indicate that a number of nurses, greater than one, are expected to be attached to the patient.

Listing 4. Nurses per patient.

```
1 structural constraint designatedPhysician(p:Patient)=
2  at p <attd> Physician; end
3 structural constraint enoughNurses(p:Patient)=
4  at p <1,attd> Nurse; end
```

3.3.3. Specifying reconfiguration constraints

Interpretation models for reconfiguration constraints have two layers. Each world represents a configuration and has an associated model for interpreting nested structural constraints. Relationships represent reconfiguration operations.

The metamodel for reconfigurations consists of a single type and a reflexive relation (see Fig. 9). The type represents configurations that an architecture may adopt, and the reflexive relation stands for a family of relations among configurations. An ordered pair of configurations is in one of such relations whenever a reconfiguration operation leads

from the first to the second configuration. Relation labels vary according to the operation and to its parameters. Then, worlds and symbols of the model correspond to configurations and reconfiguration operations, respectively.

Reconfiguration constraints exclude nominals, and redefine symbol terms to yield symbols that represent actual reconfiguration operations. Symbol terms consist of an operation name and filters, according to the syntax shown in Fig. 10, allowing us to match operation invocations (see Table 1).

The reconfiguration constraint in Listing 5 specifies restriction R1 that ensures that a nurse first checks-in and then receives patient assignments. Assume the initial configuration in Listing 2, and a nurse in a variable `n3`. A sequence of script executions will not satisfy the constraint if it allows the nurse in, and assigns the nurse to a patient, without checking the nurse in.

Listing 5. No assignment before check-in.

```
reconfiguration constraint noAssignBeforeChkIn =
forall n:Nurse.
 [true*.inew[n,_].!attach[n,_,Administration,_]*.
 attach[n,_,Patient,_] false; end
```

Reconfiguration constraint in Listing 6 specifies R2. It ensures that any sequence of scripts that unassigns a nurse leaves enough assigned. Again, assume that a sequence of scripts are triggered replacing nurse `n2` with nurse `n3` in the initial configuration, such as in Fig. 2. Since the sequence executes the unassignmet first, it fails to satisfy the constraint, because it leaves the patient with only nurse `n1` assigned. Note that the nested constraint only checks the situation of the patient whom nurse has been unassigned.

Listing 6. All unassigments leave enough nurses assigned to patients.

```
reconfiguration constraint safeUnassign =
forall p:Patient.
 [true*.detach(p,_,Nurse,_)] nest enoughNurses(p); end
```

Reconfiguration constraint in Listing 7 checks restriction R3, which ensures that any checkout takes place if the nurse has no patient assigned. It fails if, for instance, in a reconfiguration sequence such as the one in Fig. 2, nurse `n2` is checked out without unassigning the patient first.

Listing 7. No assignments before checking-out.

```
reconfiguration constraint safeCheckout =
forall n:Nurse.
  [true*.detach(n,_,Administration,_)]
  nest noAssignment(n); end
structural constraint noAssignment(n:Nurse) =
  at n not <attd> Patient; end
```

Under some circumstances it is interesting to relax structural constraints during the execution of a sequence of scripts, as long as they hold in the final configuration. For instance, when a patient enters, it may initially not have a physician and nurses assigned to him. The constraint in Listing 8 establishes that properties S1 and S2 must hold whenever a sequence of scripts cannot progress.

Listing 8. Patients safe configurations.

```
reconfiguration constraint infinite patientsSafe =
forall p:Patient.
  [true*] (<true>false implies
    nest (designatedPhysician(p) and enoughNurses(p)));
```

4. A two-layer graded hybrid logic

4.1. Syntax

Underlying the semantics of architectural descriptions and reconfigurations in Archery, is a powerful logic enabling the description of two-layer models, the reference to possible states (*i.e.* configurations) as well as to relations between them and their cardinality. As usual, such relations are denoted by modal symbols.

Formulas are called state formulas and are built upon regular formulas that define strings of modal symbols, which in turn are taken from sets given by symbol formulas. Their designation comes from their interpretation as (the) sets of states (in which they hold).

Symbol formulas, on the other hand, are interpreted as sets of modal symbols, and built as described in Definition 1. Modal symbols may take a data expression as a parameter, which is either a variable v or a function f with data expressions $e, \ldots, e$ as parameters. Symbol formulas represent sets as follows: an atomic modal symbol m is a singleton set $\{m\}$; the symbol $\top$ is the set *Sym* of all atomic modal symbols in the model; the negation $\neg\alpha$ is the complement of the set given by α; the conjunction $\alpha \wedge \alpha'$ is the intersection of the sets given by α and α'; an atomic modal symbol with a data parameter $m(e)$ is the singleton set $\{m(a)\}$, where a is the value obtained upon evaluating e; and the universal quantifier $\forall v{:}D.\alpha$ is the union of the sets obtained upon replacing v with each possible a in the set given by α.

Definition 1 (*Symbol formulas*). Let *DVar* and *Func* be disjoint sets of variables and function symbols, respectively. The set *Exp* of data expressions is recursively defined by

$$e_k \ni v_k \mid f_k(e_k, \ldots, e_k)$$

for $k \in \{0,1\}$ where $v_k \in DVar_k$ has a type D_k, and $f_k \in Func_k$ has type $D_k \times \ldots \times D_k \to D_k$. Let *Sym* be a set of atomic modal symbols. The set *MForm* of symbol formulas is recursively defined, for $k \in \{0,1\}$, by

$$\alpha_k \ni m_k \mid \top_k \mid \neg\alpha_k \mid \alpha_k \wedge \alpha_k \mid m_k(e) \mid \forall v_k{:}D_k.\alpha_k$$

where $m_k \in Sym_k$.□

For simplicity, a single data parameter is used in formulas. Multiple data parameters are obtained through projections of a composite data sort. At each level, the following operators are given by abbreviation:

$$\bot = \neg\top, \quad \alpha \vee \alpha = \neg(\neg\alpha \wedge \neg\alpha), \quad \exists v{:}D.\alpha = \neg\forall v{:}D.\neg\alpha.$$

Regular formulas are defined according to the grammar in Definition 2. At both levels, their constructs are concatenation ($\beta.\beta$), sum ($\beta+\beta$) and iteration (β^*), which are interpreted as relational composition, union and transitive reflexive closure, respectively. Additionally, the transitive closure β^+ is given by $\beta.\beta^*$.

Definition 2 (*Regular formulas*). The set *RForm* of regular formulas is recursively defined by

$$\beta_k \ni \alpha_k \mid \beta_k.\beta_k \mid \beta_k+\beta_k \mid \beta_k^*$$

for $k \in \{0,1\}$ where $\alpha_k \in MForm$.□

State formulas are formed according to the grammar in Definition 3. At both levels, a formula is either a nominal (i), a proposition (p), a negation ($\neg\varphi$), a conjunction ($\varphi \wedge \varphi$), a possibility ($\langle\beta\rangle\varphi$), a graded possibility ($\langle n,\beta\rangle\varphi$), or local reference ($@_i\varphi$) to a state I. State formulas describe two-layer models by allowing basic constructs φ_1^b at level 0.

Definition 3 (*State formulas*). A signature is an n-family of disjoint, possibly empty, sets of symbols $\Delta = (Prop_k, Nom_k)_{k \in \{0,1\}}$. The set $SForm(\Delta)$ of state formulas is recursively defined as

$$\varphi_0 \ni \varphi_1^b \mid i_0 \mid p_0 \mid \neg\varphi_0 \mid \varphi_0 \wedge \varphi_0 \mid \langle\beta\rangle_0\,\varphi_0 \mid \langle n,\beta\rangle_0\,\varphi_0 \mid @_{i_0}\varphi_0$$

and

$$\varphi_1^b \ni p_1 \mid i_1 \mid \langle\beta\rangle_1\,\varphi_1 \mid \langle n,\beta\rangle_1\,\varphi_1 \mid @_{i_1}\varphi_1$$
$$\varphi_1 \ni i_1 \mid p_1 \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_1 \mid \langle\beta\rangle_1\,\varphi_1 \mid \langle n,\beta\rangle_1\,\varphi_1 \mid @_{i_1}\varphi_1$$

where $p_k \in Prop_k$, and $i_k \in Nom_k$ for $k \in \{0,1\}$.□

Additionally, at both levels, the following constructs are defined by abbreviation:

$$\bot = \neg\top, \qquad \varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$$
$$\varphi_1 \to \varphi_2 = \neg\varphi_1 \vee \varphi_2, \quad \varphi_1 \leftrightarrow \varphi_2 = \varphi_1 \to \varphi_2 \wedge \varphi_2 \to \varphi_1$$
$$[\beta]\varphi = \neg\langle\beta\rangle\neg\varphi, \qquad \neg\langle n,\beta\rangle\varphi = [n,\beta]\neg\varphi$$

4.2. Semantics

Definition 4 (*2-layer model*). A 2-layer model $\mathfrak{M} \in \mathsf{Model}(\Delta)$ is a tuple $\mathfrak{M} = (\mathfrak{M}_k)_{k \in \{0,1\}} = (W_k, Q_k, R_k, V_k)_{k \in \{0,1\}}$ recursively defined as follows:

- $(W_k)_{k \in \{0,1\}}$ are disjoint sets;
- $(Q_k)_{k \in \{0,1\}}$ are predicates with $Q_0 \subseteq W_0$ and $Q_1 \subseteq W_0 \times W_1$ such that

$$W_1 = \{w_1 : \exists w_0 \in W_0.Q_0(w_0) \wedge Q_1(w_0, w_1)\};$$

- $(R_k: Mod_k \to Q_k \times Q_k)_{k \in \{0,1\}}$ is a pair of functions; and
- $(V_k^{Prop}, V_k^{Nom})_{k \in \{0,1\}}$ are pairs of functions:
  - $V_0^{Prop}: Prop_0 \to \mathcal{P}(W_0)$ and $V_1^{Prop}: Prop_1 \times Q_0 \to \mathcal{P}(W_1)$, and
  - $V_0^{Nom}: Nom_0 \to W_0$ and $V_1^{Nom}: Nom_1 \to W_1$.

A 2-layer model $\mathfrak{M}$ is said to be hierarchical if

$$R[s]_0 = \{(w_0, w_0'): R[s]_1((w_0, w_1), (w_0', w_1)) \text{ for some } w_1 \text{ and } w_1'\}$$

for any $s \in Mod_1$. A hybrid model $\mathfrak{M} = (W, R, V)$ is a tuple where $W = W_0$, $R = R_0$, and $V = V_0$.□

Let us fix the following notation: expression $\mathfrak{m}[d \mapsto r]$ denotes a map $\mathfrak{m}'$ in which $\mathfrak{m}'(d') = \mathfrak{m}(d')$ for all $d' \neq d$ and $\mathfrak{m}'(d) = r$ otherwise; $supp(\mathfrak{m})$ denotes the set of values mapped by $\mathfrak{m}$ and is called its *support*.

Definition 5 (*Satisfaction*). Let $\mathbb{D}$ denote the set of values of a variable v of type D, and let $\mathfrak{v} = (\mathfrak{v}_k)_{k \in \{0,1\}}$ be a pair of data environments. The value of a data expression is given by a pair of interpretation functions $(\llbracket\ \rrbracket_{k,\mathfrak{v}_k}: Exp \to \mathbb{D}_k)_{k \in \{0,1\}}$ given by

$$\llbracket v_k \rrbracket_{k,\mathfrak{v}_k} \triangleq \mathfrak{v}_k(v_k), \quad \llbracket f_k(e_k, \dots e_k) \rrbracket_{k,\mathfrak{v}_k} \triangleq f_k(\llbracket e_k \rrbracket_{k,\mathfrak{v}_k}, \dots, \llbracket e_k \rrbracket_{k,\mathfrak{v}_k}),$$

parametric on a data environment $\mathfrak{v}_k: DVar_k \to \mathbb{D}_k$ that assigns a value to a variable, and such that $var(e_k) \subseteq supp(\mathfrak{v}_k)$, where $var(e_k)$ denotes the variables occurring in an expression e_k.

The interpretation of a symbol formula is given by a pair of functions $(\llbracket\ \rrbracket_{k,\mathfrak{v}_k}: MForm \to Mod_k)_{k \in \{0,1\}}$ defined inductively as

$$\begin{aligned}
\llbracket m_k(e_k) \rrbracket_{k,\mathfrak{v}_k} &\triangleq \{m_k(\llbracket e_k \rrbracket_{k,\mathfrak{v}_k})\}, \qquad \llbracket \top_k \rrbracket_{k,\mathfrak{v}_k} \triangleq Mod_k \\
\llbracket \neg\alpha_k \rrbracket_{k,\mathfrak{v}_k} &\triangleq Mod_k \setminus \llbracket \alpha_k \rrbracket_{k,\mathfrak{v}_k}, \quad \llbracket \alpha_k \wedge \alpha_k \rrbracket_{k,\mathfrak{v}_k} \triangleq \llbracket \alpha_k \rrbracket_{k,\mathfrak{v}_k} \cap \llbracket \alpha_k \rrbracket_{k,\mathfrak{v}_k} \\
\llbracket \forall v_k{:}D_k.\alpha_k \rrbracket_{k,\mathfrak{v}_k} &\triangleq \bigcap_{a:\mathbb{D}_k} \llbracket \alpha_k \rrbracket_{\mathfrak{v}_k'},
\end{aligned}$$

where $\mathfrak{v}_k' = \mathfrak{v}_k[v \mapsto a]$.

The interpretation of a regular formula is given by a pair of functions $(\Vert\ \Vert_{k,\mathfrak{v}_k}: RForm \to Rel_k)_{k \in \{0,1\}}$ defined inductively as follows:

$$\begin{aligned}
\Vert\alpha\Vert_{k,\mathfrak{v}_k} &\triangleq \{(w, w'): (w, w') \in R[s] \text{ for some } s \in \llbracket \alpha \rrbracket_{k,\mathfrak{v}_k}\} \\
\Vert\beta_k.\beta_k'\Vert_{k,\mathfrak{v}_k} &\triangleq \Vert\beta_k\Vert_{k,\mathfrak{v}_k} \bigcirc \Vert\beta_k'\Vert_{k,\mathfrak{v}_k} \\
\Vert\beta_k + \beta_k\Vert_{k,\mathfrak{v}_k} &\triangleq \Vert\beta_k\Vert_{k,\mathfrak{v}_k} \cup \Vert\beta_k\Vert_{k,\mathfrak{v}_k} \\
\Vert\beta^*\Vert_{k,\mathfrak{v}_k} &\triangleq \Vert\beta\Vert^*_{k,\mathfrak{v}_k}
\end{aligned}$$

where Rel_k is the set of all relations in $Q_k \times Q_k$, $\Vert\beta_1\Vert_{k,\mathfrak{v}_k} \circ \Vert\beta_2\Vert_{k,\mathfrak{v}_k}$ is a concatenation, and $\Vert\beta\Vert^*_{k,\mathfrak{v}_k}$ is a transitive reflexive closure.

Let $\mathfrak{M}$ be a 2-layer model. The satisfaction of state formulas w.r.t. a data environment $\mathfrak{v}$ is given by a pair of relations $(\vDash_k)_{k \in \{0,1\}}$ defined as follows:

$$\mathfrak{M}_0, \mathfrak{v}_0, w_0 \vDash_0 \varphi_1^b \quad \text{iff } \mathfrak{M}_1, \mathfrak{v}_1, w_1 \vDash_1 \varphi_1^b \text{ and } Q_1(w_0, w_1) \text{ for some } w_1 \in W_1 \tag{1}$$

$$\mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k i_k \quad \text{iff } w_k = V_k^{Nom}(i_k) \tag{2}$$

$$\mathfrak{M}_0, \mathfrak{v}_0, w_0 \vDash_0 p_0 \quad \text{iff } w_0 \in V_0^{Prop}(p_0) \tag{3}$$

$$\mathfrak{M}_1, \mathfrak{v}_1, w_1 \vDash_1 p_1 \quad \text{iff } w_1 \in V_1^{Prop}(p_1, w_0) \text{ and } Q_1(w_0, w_1) \tag{4}$$

$$\mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k \varphi_k \wedge \varphi_k' \quad \text{iff } \mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k \varphi_k \text{ and } \mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k \varphi_k' \tag{5}$$

$$\mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k \neg\varphi_k \quad \text{iff it is false that } \mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k \varphi_k \tag{6}$$

$$\begin{aligned}
\mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k \langle\beta\rangle_k\varphi_k \quad &\text{iff } \mathfrak{M}_k, \mathfrak{v}_k, v_k \vDash_k \varphi_k \\
&\text{for some } v_r \in W_r, r \in \{0, \dots, k\} \\
&\text{such that } ((w_0, \dots, w_k), (v_0, \dots, v_k)) \in \Vert\beta\Vert_{k,\mathfrak{v}_k}
\end{aligned} \tag{7}$$

$$\begin{aligned}
\mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k \langle n, \beta\rangle_k\varphi_k \quad &\text{iff } n < |\{v_k: \mathfrak{M}_k, \mathfrak{v}_k, v_k \vDash_k \varphi_k \\
&\text{for some } v_r \in W_r, r \in \{0, \dots, k\} \\
&\text{such that } ((w_0, \dots, w_k), (v_0, \dots, v_k)) \in \Vert\beta\Vert_{k,\mathfrak{v}_k}\}|
\end{aligned} \tag{8}$$

$$\mathfrak{M}_k, \mathfrak{v}_k, w_k \vDash_k @_{i_k}\varphi_k \quad \text{iff } \mathfrak{M}_k, \mathfrak{v}_k, V_k^{Nom}(i_k) \vDash_k \varphi_k \tag{9}$$

for $k \in \{0, 1\}$, and each $w_r \in W_r, r \in \{0, \dots, k\}$ with $Q_k(w_0, \dots, w_k)$.□

4.3. Bisimulation and invariance

Bisimulation offers a basic, actually quite strong, form of equivalence between models of the logic proposed in this section, and consequently between Archery configurations. This section introduces suitable notions of bisimulation and refinement and explores their relationship with logical satisfaction. This leads to a Hennessy–Milner like theorem of broad relevance. We start by recalling what a bisimulation is in the standard (one layer) hybrid modal logic:

Definition 6 (*Bisimulation*). Let $\mathfrak{M}=(W,R,V)$ and $\mathfrak{M}'=(W',R',V')$ be two hybrid models over the same signature. A *bisimulation* between $\mathfrak{M}$ and $\mathfrak{M}'$ consists of a relation

$$Z \subseteq W \times W'$$

such that

(Nom) for any $i \in Nom$, $V(i)\, Z\, V'(i)$,

and for any $w \in W$ and $w' \in W'$,

$$w\, Z\, w'$$

implies that

(Atoms) for any $\sigma \in Prop \cup NOM$, $w \in V(\sigma)$ iff $w' \in V'(\sigma)$;
(n-Zig) for any positive n, for each $m \in Sym$ and for any n distinct $v_k \in W$, $k \in 0,\ldots,n$ such $(w, v_k) \in R[m]$, there are n distinct $v'_k \in W'$, $k \in 1,\ldots,n$, such that $(w', v'_k) \in R'[m]$ with $v_k\, Z\, v'_k$ for any $k \in 0,\ldots,n$.
(n-Zag) for any positive n, for each $m \in Sym$ and for any n distinct $v'_k \in W'$, $k \in 0,\ldots,n$ such $(w', v'_k) \in R'[m]$, there are n distinct $v_k \in W$, $k \in 1,\ldots,n$, such that $(w, v_k) \in R[m]$ with $v_k\, Z\, v'_k$ for any $k \in 0,\ldots,n$.

Definition 7 (*Hierarchical bisimulation*). A *hierarchical bisimulation* between two hierarchical models $\mathfrak{M}=(W_k,Q_k,R_k,V_k)_{k\in\{0,1\}}$ and $\mathfrak{M}'=(W'_k,Q'_k,R'_k,V'_k)_{k\in\{0,1\}}$ consists of a family of relations $(Z_k \subseteq Q_k \times Q'_k)_{k\in\{0,1\}}$, such that

- for any $k \in \{0,1\}$, Z_k is a bisimulation,
- for any w_0, w'_0 such that $w_0\, Z_0\, w'_0$,
  (i) and for each w_1 such that and $Q_1(w_0,w_1)$ there is a w'_1 such that $(w_0,w_1)Z_1(w'_0,w'_1)$ (10)
  (ii) and each w'_1 such that and $Q'_1(w'_0,w'_1)$ there is a w_1 such that (10)

The following result establishes bisimulation invariance with respect to the proposed logic.

Theorem 1. *Let Z be a bisimulation between the hierarchical models $\mathfrak{M}=(W_k,Q_k,R_k,V_k)_{k\in\{0,1\}}$ and $\mathfrak{M}'=(W'_k,Q'_k,R'_k,V'_k)_{k\in\{0,1\}}$, and $w \in W, w' \in W'$ two states such that $(w_0,w_1)Z_1(w'_0,w'_1)$. Then, for any data environment $\mathfrak{v}$ and for any formula φ,* we *have that*

$$\mathfrak{M}_1,\mathfrak{v}_1,w_1 \vDash_1 \varphi \quad \textit{iff}\ \mathfrak{M}'_1,\mathfrak{v}_1,w'_1 \vDash_1 \varphi$$

Proof. We start observing that, since both Z_1 and Z_0 are bisimulations, the invariance of 0-sentences across Z_0 and the invariance of 1-sentences across Z_1, with the exception of sentences φ^b_1, can be proved as in the standard hybrid modal logic with graded modalities (*e.g.* [13,14]), by induction over the structure of the sentences. The preservation of the latter comes as follows:

$$\begin{aligned}
&\mathfrak{M}_1,\mathfrak{v}_1,w_1 \vDash_1 \varphi^b_1\\
&\equiv \{\text{defn. of } \vDash_k\}\\
&\quad \mathfrak{M}_0,\mathfrak{v}_0,w_0 \vDash_0 \varphi^b_1 \text{ for some } w_1 \text{ such that } Q_1(w_0,w_1)\\
&\equiv \{\text{step } \star\}\\
&\quad \mathfrak{M}'_0,\mathfrak{v}_0,w'_0 \vDash_0 \varphi^b_1 \text{ for some } w'_1 \text{ such that } Q'_1(w'_0,w'_1)\\
&\equiv \{\text{defn. of } \vDash_k\}\\
&\quad \mathfrak{M}'_1,\mathfrak{v}_1,w'_1 \vDash_1 \varphi^b_1
\end{aligned}$$

For step $\star$ note that items (i) and (ii) in Definition 7 assure the existence of $w_1 \in W_1$ and $w'_1 \in W'_1$ such that $(w_0,w_1)Z_1(w'_0,w'_1)$. Hence, the equivalence is justified by the invariance result for of hybrid logics with graded modalities.□

The existence of a Hennessy–Milner like theorems for the logic is now discussed. For this, let us consider that a hierarchical model $\mathfrak{M}$ is *image-finite* when each of its outer and inner accessibility relations are image-finite, *i.e.*, for each $w \in W_k$ and $m \in Sym_k$, the sets $\{w' : (w, w') \in R[m]_k\}$, $k \in \{0, 1\}$, are finite.

Actually, for any two image-finite models $\mathfrak{M}$ and $\mathfrak{M}'$ and for any $w \in W$ and $w' \in W'$ we can prove the equivalence of statements:

(i) $\mathfrak{M}_1, w_1 \vDash_1 \varphi$ iff $\mathfrak{M}'_1, w'_1 \vDash_1 \varphi$
(ii) There is a bisimulation $Z \subseteq W \times W'$ between $\mathfrak{M}$ and $\mathfrak{M}'$.

whenever φ is free graded modalities. To discuss under which conditions the implication (i) $\Rightarrow$ (ii) (*i.e.* the converse of Theorem 1) holds, let us consider the relation $\mathfrak{M}_1, w_1 \vDash_1 \varphi$ iff $\mathfrak{M}'_1, w'_1 \vDash_1 \varphi$

$$Z := \{(w, w') \in W \times W' \mid \mathfrak{M}_1, w_1 \vDash_1 \varphi \text{ iff } \mathfrak{M}'_1, w'_1 \vDash_1 \varphi\}$$

Clearly Z satisfies the (Nom) and (Atoms) conditions in the definition of bisimulation. Let $u_k \in W$, $k = 1, \ldots, n$ such that $(w, u_k) \in R[m]$ for any $k = 1, \ldots, n$. Suppose also that there are not n distinct $u'_k \in W'$, $k = 1, \ldots, n$, such that $(w', u'_k) \in R[m]$ and $u_k \, Z \, u'_k$. By hypothesis, $w \in [\![\langle n, m \rangle \top]\!]_{\mathfrak{M},\mathfrak{v}}$. By (i), this entails $w' \in [\![\langle n, m \rangle \top]\!]_{\mathfrak{M}',\mathfrak{v}}$. Therefore, we conclude that $u_k \, Z \, u'_k$ does not hold. Hence, there is at least a formula ψ_k such that $u_k \in [\![\psi_k]\!]_{\mathfrak{M},\mathfrak{v}}$ and $u'_k \notin [\![\psi_k]\!]_{\mathfrak{M}',\mathfrak{v}}$. Restricting the modalities to the non-graded case, the verification of (ZIG) comes from observing that $w \in [\![\langle m \rangle \bigvee_{k \in \{1,\ldots,n\}} \psi_k]\!]_{\mathfrak{M},\mathfrak{v}}$ leads to a contradiction (because $w' \in [\![\langle m \rangle \bigvee_{k \in \{1,\ldots,n\}} \psi_k]\!]_{\mathfrak{M}',\mathfrak{v}}$ and $u'_k \notin [\![\psi_k]\!]_{\mathfrak{M}',\mathfrak{v}}$). However, in the graded case, we cannot find any formula $\langle n, m \rangle \Phi$ to achieve such kind of contradiction.

Finally we observe that the recent work [15] presents a description logic with graded modalities that is endowed with a bisimulation notion satisfying a Henessy–Milner property. Its ZIG-ZAG correspondence, however, is established by a bijective relation, which is a very strong condition in view of our purposes.

4.4. Refinement

We introduce, in this section, a property preserving relation between hierarchical models.

Definition 8. Let $\mathfrak{M} = (W, R, V)$ and $\mathfrak{M}' = (W', R', V')$ be two hybrid models over the same signature. A *simulation* between $\mathfrak{M}$ and $\mathfrak{M}'$ consists of a relation $S \subseteq W \times W'$ such that

(Nom) for any $i \in Nom$, $V(i) \, Z \, V'(i)$,

and for any $w \in W$ and $w' \in W'$, $w \, Z \, w'$ implies that :

(Atoms) for any $\sigma \in Prop \cup NOM$, if $w \in V(\sigma)$ then $w' \in V'(\sigma)$;
(n-Zig) for any positive n, for each $m \in Sym$ and for any n distinct $v_k \in W$, $k \in 0, \ldots, n$ such $(w, v_k) \in R[m]$, there are n distinct $v'_k \in W'$, $k \in 1, \ldots, n$, such that $(w', v'_k) \in R'[m]$ with $v_k \, S \, v'_k$ for any $k \in 0, \ldots, n$.

Definition 9 (*Hierarchical refinement*). A *hierarchical refinement* between two hierarchical models $\mathfrak{M} = (W_k, Q_k, R_k, V_k)_{k \in \{0,1\}}$ and $\mathfrak{M}' = (W'_k, Q'_k, R'_k, V'_k)_{k \in \{0,1\}}$ consists of a family of relations $(S_k \subseteq Q_k \times Q'_k)_{k \in \{0,1\}}$, such that

- for any $k \in \{0, 1\}$, S_k is a simulation,
- for any w_0, w'_0 such that $w_0 \, S_0 \, w'_0$, and for each w_1 such that and $Q_1(w_0, w_1)$ there is a w'_1 such that $(w_0, w_1) S_1 (w'_0, w'_1)$.

Next theorem establishes the preservation of properties over across. The proof is omitted since it can be directly derived from the one of Theorem 1. Note however that, since the refinement relation just imposes the directional preservation of the propositions and atomic formulas, the preservation of negation, and consequently of boxes, is lost. Hence,

Theorem 2. *Let S be a refinement relation between the hierarchical models $\mathfrak{M} = (W_k, Q_k, R_k, V_k)_{k \in \{0,1\}}$ and $\mathfrak{M}' = (W'_k, Q'_k, R'_k, V'_k)_{k \in \{0,1\}}$ and $w \in W$, $w' \in W'$ two states such that $(w_0, w_1) Z_1 (w'_0, w'_1)$. Then, for any data environment $\mathfrak{v}$ and for any formula φ without diamonds and negations, we have that*

$$\mathfrak{M}_1, \mathfrak{v}_1, w_1 \vDash_1 \varphi \text{ implies } \mathfrak{M}'_1, \mathfrak{v}_1, w'_1 \vDash_1 \varphi.$$

5. Verifying hierarchical constraints

Verifying a constraint over a specification requires deriving an interpretation model $\mathfrak{M}$ and then translating the constraint into the corresponding logic. The derivation of interpretation models for structural constraints is detailed in Appendix A. On the other hand, the derivation of two-layer models can be found in [10],

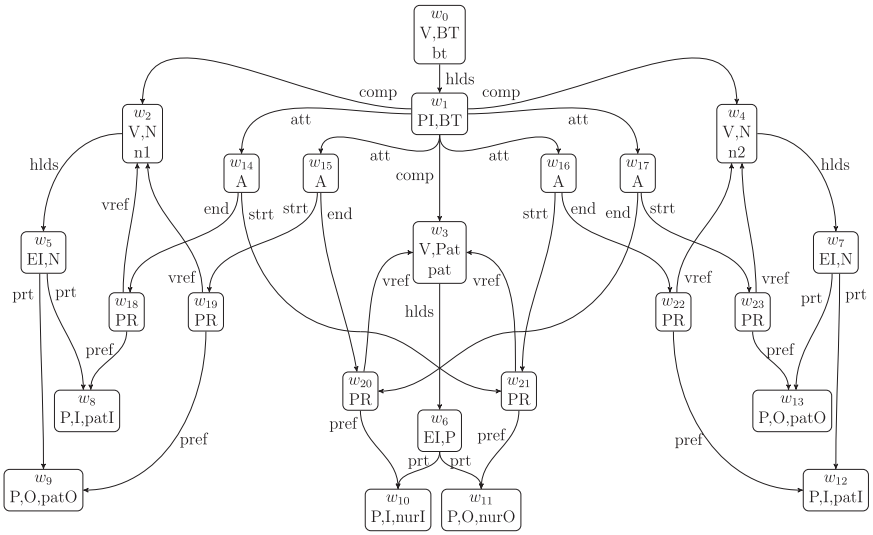

Fig. 11. Partial model for the initial configuration.

Reconfiguration constraints become formulas of a 2-layer modal logic. Translations differ on how variables are treated. In structural constraints, variables are bound to nominals. Then, the meaning of **forall** x: TYPEID is the conjunction of formulas **at** x F $[x/i]$, for each $i \in Nom_{var:TYPEID}$, where $[x/i]$ denotes the substitution of x by i in F. Dually, the meaning of **exists** x: TYPEID F is a disjunction of formulas **at** x F $[x/i]$, for each $i \in Nom_{TYPEID}$. Note that in reconfiguration constraints, variables become data variables. The types that a pattern defines become data sorts, configuration variables become values of such sorts, and ports become values of a port data sort. Appendix B provides the precise definition of the translations.

To illustrate our approach, we verify now the constraint `safeUnassign` (in Listing 6) over the sequence of script executions that replaces a nurse, shown in Fig. 2.

First, it is shown that the initial configuration satisfies the nested structural constraint `enoughNurses` (see Listing 4). The result of translating the Archery specification of the initial configuration into an interpretation model is shown in Fig. 11. It is partial since instances of administration and physician are omitted. In addition, names (see the metamodel) and their relationships are also dropped. Each node in the graph represents a world and includes an identifier in the first line; the satisfied propositions in the second line; and the satisfied nominals in the third line. A short code is used for the propositions that depend on the pattern: N (Nurse) and Pat (Patient). A short symbol is used to avoid using the longer symbols of propositions: V (Variable), PI (PatternInstance), EI (ElementInstance), P (Port), I (In), O (Out), A (Attachment), R (Renaming), PR (PortReference), Act (Action), and N (Name).

The resulting model is simplified into the model shown in Fig. 12(a), by considering the relation `attd`, which is the only one present in the formula. It does not show worlds representing instances in variables, ports, and attachments.

The nested constraint is translated into formula

$$@pat\ \langle 1, attd\rangle\ Nurse.$$

with the parameter bound to `pat`, the unique patient in the initial configuration. Its verification proceeds as follows:

$$\begin{aligned}
&@pat\ \langle 1, attd\rangle\ Nurse\\
&= \{\text{by (4)}\}\\
&\quad @pat\ \langle 1, attd\rangle\{w_2, w_4\}\\
&= \{\text{by (8)}\}\\
&\quad @pat\ \{w_3\}\\
&= \{\text{by (9)}\}\\
&\quad W
\end{aligned}$$

The formula is satisfied by the initial configuration.

In contrast, configuration c_5 in Fig. 2 does not. Nurse `n2` was unassigned before assigning `n3`. The derived model is shown in Fig. 12(b) and the verification is as follows:

$$\begin{aligned}
&@pat\ \langle 1, attd\rangle\ Nurse\\
&= \{\text{by (4)}\}\\
&\quad @pat\ \langle 1, attd\rangle\{w_2, w_4, w_5\}\\
&= \{\text{by (8)}\}
\end{aligned}$$

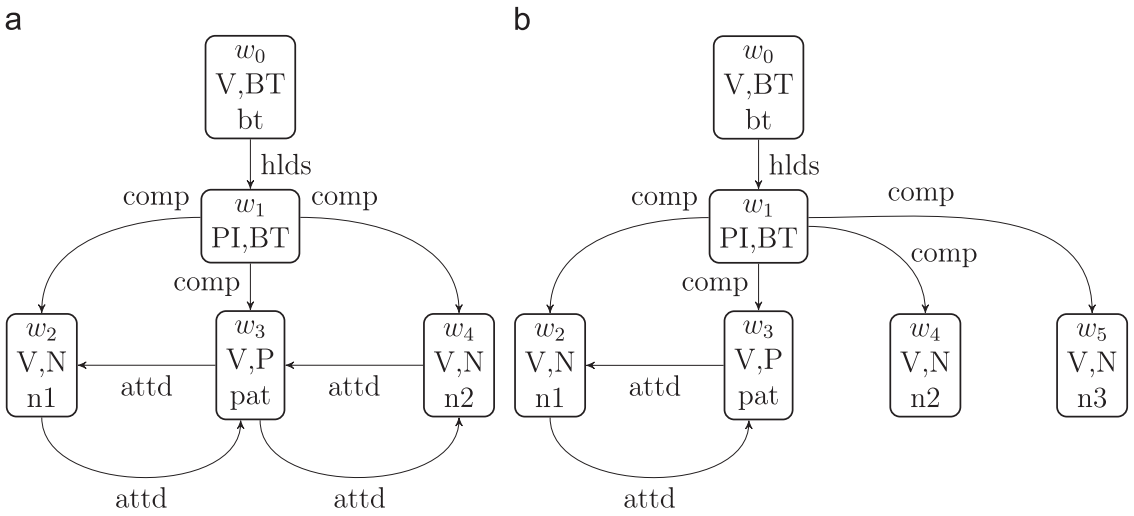

Fig. 12. Partial models. (a) Initial configuration. (b) Configuration upon unassignment.

$$
\begin{aligned}
& @pat\ \varnothing \\
&= \{\text{by (9)}\} \\
& \varnothing
\end{aligned}
$$

The model for the sequence of script executions, which is referred as subs0 in the sequel, is partially shown in Fig. 13. Its first level consists of the primitives that result from the execution of subs0, and it shows the nested models for the initial configuration c_0 and for the configuration upon the unassignment of the nurse c_5.

The translation of the reconfiguration constraint `safeUnassign`, shown in Listing 6, yields formula

$$[\top^*.detach(pat, nurO, n2, patI)]\ \psi(pat),$$

where ψ stands for the formula of constraint `enoughNurses`. The symbol term is replaced by the only symbol that matches the criteria within the necessity operator. Likewise, the parameter of the nested constraint is fixed to the unique variable of type Patient in the successive configurations. The formula is verified as follows:

$$
\begin{aligned}
& [\top^*.detach(pat, nurO, n2, patI)]\ \psi(pat) \\
&= \{\text{by } [\beta]\varphi = \neg\langle\beta\rangle\neg\varphi\} \\
& \quad \neg\langle\top^*.detach(pat, nurO, n2, patI)\rangle\ \neg\psi(pat) \\
&= \{\text{by (1)}\} \\
& \quad \neg\langle\top^*.detach(pat, nurO, n2, patI)\rangle\ \neg(W\backslash\{c_5\}) \\
&= \{\text{by (6)}\} \\
& \quad \neg\langle\top^*.detach(pat, nurO, n2, patI)\rangle\ \{c_5\} \\
&= \{\text{by (7) and } \langle\beta.\beta'\rangle = \langle\beta\rangle\langle\beta'\rangle\} \\
& \quad \neg\langle\top^*\rangle\ \{c_4\} \\
&= \{\text{by (7) and } \langle\beta.\beta'\rangle = \langle\beta\rangle\langle\beta'\rangle \text{ four times}\} \\
& \quad \neg\{c_0\} \\
&= \{\text{set complement}\} \\
& \quad W\backslash\{c_0\}
\end{aligned}
$$

Since the result excludes the initial configuration, reconfigurations subs0 fail to satisfy the constraint. On the other hand, the reconfigurations in Fig. 14(a), subs1 in the sequel, satisfy it, as they assign the patient to `n3` before unassigning `n2`. Reconfigurations subs0 and subs1 are not bisimilar, and none of them are refinement of the other.

Consider now reconfiguration subs2 in Fig. 14(b). It proceeds as subs1, but offers an optional sequence of operations that the configuration manager can perform. Nurse `n4` is let in, and checked in by such additional operations. Reconfigurations subs2 and subs1 are not bisimilar, but subs2 is a refinement of subs1. The relation that satisfies Definition 9 contains pairs (c_j, c'_j) where c_j and c'_j are in the interpretation models of subs1 and subs2, respectively, and $j \in \{0\ldots10\}$. Since subs2 is a refinement of subs1, Theorem 2 can be applied to avoid verifying constraints already valid in subs1. However, since the constraint we are studying includes diamonds, it is out of the scope of such a theorem, and the verification on subs2 is still required.

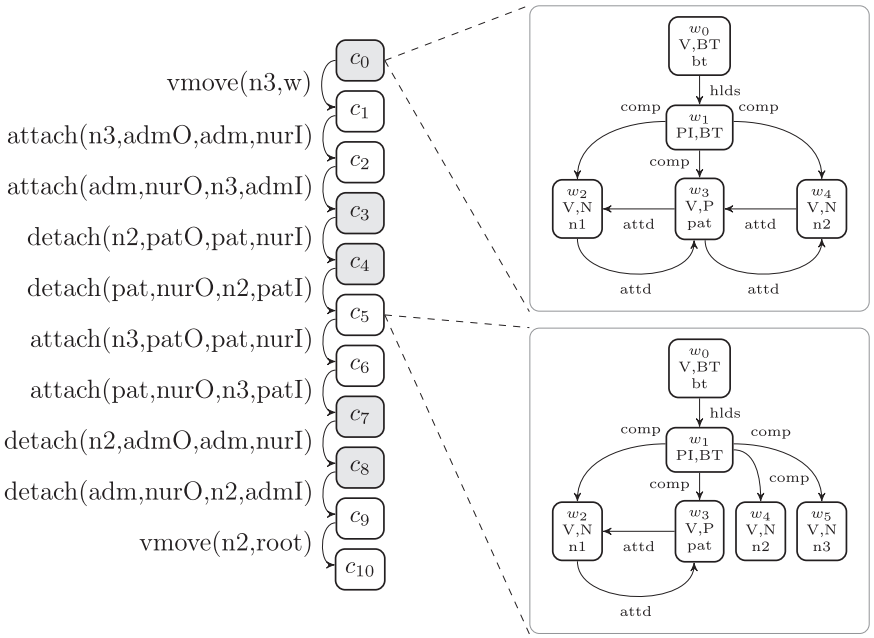

Fig. 13. Partial 2-layer model (subs0).

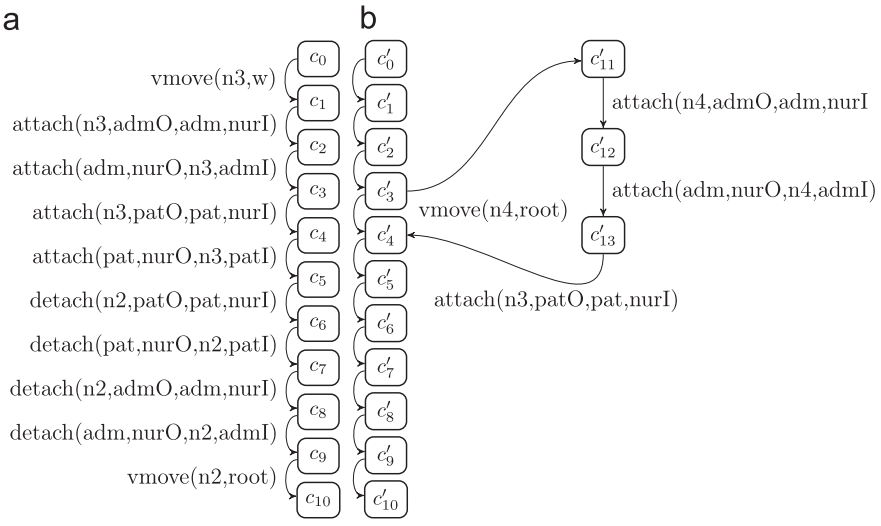

Fig. 14. Correct reconfigurations. (a) Nurse substitution (subs1). (b) Substitution and optional check in (subs2).

6. Related work

An *architectural description language* ADL provides a number of typical abstractions to model software architectures. Reference [2] identifies components, connectors, and configurations as essential elements in such a description. In addition, the concept of architectural pattern (or style) [16,17] facilitates the development of specifications since it allows abstracting recurring forms. ACME [18], Darwin [19–21], Wright [22,23], and ADR [24] are among the languages that support these abstractions.

ADLs with formal semantics provide a sound foundation for the tool-supported development of dynamically reconfigurable architectural models. It is possible to distinguish among languages precisely defined upon process algebras and graphical theories [25]. While Darwin [19] and Wright [23] are examples of the former, ADR [24] combines both. Archery [4] models the behavioural part of software architectures with process algebras [26], and the structural part with bigraphical reactive systems [27].

These languages represent and analyse dynamic reconfigurable architectures in different ways [19,23,28]. They enforce architectural principles in a pattern either *by construction* or *by restriction*. The former requires defining pattern-specific

reconfiguration operations that can only produce correct configurations according to design principles [29]. Then, design decisions are left implicit in these operations. ADR [24] uses this mechanism. The latter approach requires the explicit specification of constraints that prevent generic (re)configuration operations leading to incorrect configurations. Archery follows this approach. ADLs Darwin [20] and ACME [30] also follow this approach by providing a translation into the Alloy language [31], which is based on a first-order relational logic and is supported by a bounded model checker implemented upon a SAT solver.

The approach proposed in this paper has connections to the one documented in [32], which is based on a layered approach for addressing dynamic reconfiguration. It includes a layer where properties about configurations are specified using first order logic, a layer where reconfiguration events can be related to configuration properties, another one that restricts the way configuration properties appear in traces, and, finally, a layer that describes valid combinations of reconfiguration events and traces of configurations, specified in a linear temporal logic.

An ADL for reasoning about behavioural constraints, expressed in a combination of first order and temporal logic, in the presence of dynamic reconfiguration is proposed in [33]. The concept of an architectural reconfiguration contract [34] relates the behaviour of a system and its dynamic reconfigurations. It establishes at which states a configuration can be replaced by another one, and the states at which the new configuration can be safely left. Archery's support for this concept is part of ongoing work.

7. Conclusions and future work

This paper proposes the use of hierarchical constraints to ensure that architectural reconfigurations on a system proceed as expected. Resulting configurations respect system's architectural principles, as inherited from the specific architectural pattern adopted or originally fixed by the software architect. The Archery language is used to specify architectures, reconfigurations and constraints. Translations into a two-layer graded hybrid logic are presented, which enable the formal verification of constraints. The approach is illustrated with the verification of architectural principles that must be respected by (re)configurations of a service architecture for the blood transfusion procedure.

7.1. Future and ongoing work

Ongoing work is concerned with the specification of nested behavioural constraints, the usage of constraints for triggering reconfiguration scripts, and the provision of tool-support for a fragment of the language that excludes nesting, *i.e.*, constraints that can be translated into a graded hybrid logic. Future work, on the other hand, includes the study of tool-support for the whole constraint language, and the development of a comprehensive case study in the context of electronic government.

Acknowledgements

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through FCT, the Portuguese Foundation for Science and Technology, within project FCOMP-01-0124-FEDER-028923. The second author is also supported by the FCT grant SFRH/BPD/103004/2014.

Appendix A. Deriving models for structural constraints

As a convention, the expression $\mathfrak{m}[l{\rightrightarrows}u]$ denotes $\mathfrak{m}[l{\mapsto}\mathfrak{m}(l)\cup u]$ if u is a set, and $\mathfrak{m}[l{\mapsto}\mathfrak{m}(l)\cup\{u\}]$ otherwise.

Definition 10 (*Model derivation for structural constraints*). Let $\mathcal{W}\colon Part\to W$ be a function that yields a world that represents a participant in an architecture, where $Part = Var \uplus ElemInst \uplus PatInst \uplus Att \uplus PortRef \uplus Ren \uplus Id \uplus Act$, let Model_S be the set of models for $\mathfrak{M}_S$, and let $\mathcal{N}\colon TYPE\to\mathcal{P}(Nom)$ be a map that yields, for a type defined by a pattern, the nominals representing variables of such type in a given architecture. The derivation is given by a function $\mathcal{I}\colon Var\times W\times \mathsf{Model}_S\times\mathcal{N}\to\mathsf{Model}_S\times\mathcal{N}$ which is inductively defined as follows:

$$
\begin{aligned}
&\mathcal{I}(\mathrm{v} = (\mathrm{id\ type\ inst}), w_v, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \quad \mathcal{I}(\mathrm{inst}, w_i, (W\cup\{w_v,w_i\}, \mathcal{R}[\mathrm{hlds}{\rightrightarrows}(w_v,w_i)], V[\mathrm{V}{\rightrightarrows}w_v][\mathrm{type}{\rightrightarrows}w_v]),\\
&\quad \mathcal{N}[\mathrm{type}{\rightrightarrows}\mathrm{id}]),\ \text{where } w_v = \mathcal{W}(\mathrm{v}) \text{ and } w_i = \mathcal{W}(\mathrm{inst})\\
&\mathcal{I}(\mathrm{inst} = (\mathrm{type\ ports\ acts}), w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \mathcal{I}(\mathrm{ports\ acts}, w_i, (W,\mathcal{R},V[\mathrm{type}{\rightrightarrows}w_i][\mathrm{EI}{\rightrightarrows}w_i]),\mathcal{N})\\
&\mathcal{I}(\mathrm{port} = (\mathrm{id\ dir\ ptype})\ \mathrm{ports\ acts}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \mathcal{I}(\mathrm{ports\ acts}, w_i, (W\cup\{w_{\mathrm{port}}, w_{\mathrm{id}}\},\\
&\quad \mathcal{R}[\mathrm{prt}{\rightrightarrows}(w_i,w_{\mathrm{port}})][\mathrm{name}{\rightrightarrows}(w_{\mathrm{port}},w_{\mathrm{id}})],\\
&\quad V[\mathrm{P}{\rightrightarrows}w_{\mathrm{port}}][\mathrm{dir}{\rightrightarrows}w_{\mathrm{port}}][\mathrm{id}{\rightrightarrows}w_{\mathrm{port}}][\mathrm{ptype}{\rightrightarrows}w_{\mathrm{port}}][\mathrm{N}{\rightrightarrows}w_{\mathrm{id}}]),\mathcal{N})
\end{aligned}
$$

$$
\begin{aligned}
&\text{where } w_{\text{port}} = \mathcal{W}(\text{port}) \text{ and } w_{\text{id}} = \mathcal{W}(\text{id})\\
&\mathcal{I}([\,]\ \text{acts}, w_i, (W,\mathcal{R},V),\mathcal{N}) = \mathcal{I}(\text{acts}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\mathcal{I}(\text{act} = \text{id acts}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \mathcal{I}(\text{acts}, w_i, (W \cup \{w_{\text{act}}, w_{\text{id}}\}, \mathcal{R}[\text{act}{\rightrightarrows}(w_i, w_{\text{act}})][\text{name}{\rightrightarrows}(w_{\text{act}}, w_{\text{id}})],\\
&\quad V[\text{Act}{\rightrightarrows}w_{\text{act}}][\text{N}{\rightrightarrows}w_{\text{id}}]), \mathcal{N}),\ \text{where } w_{\text{act}} = \mathcal{W}(\text{act})\\
&\mathcal{I}([\,], w_i, (W,\mathcal{R},V),\mathcal{N}) = ((W,\mathcal{R},V),\mathcal{N})\\
&\mathcal{I}(\text{inst} = (\text{type vs atts rens}), w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \mathcal{I}(\text{vs atts rens}, w_i, (W,\mathcal{R},V[\text{type}{\rightrightarrows}w_i][\text{PI}{\rightrightarrows}w_i]),\mathcal{N})\\
&\mathcal{I}(\text{v vs atts rens}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \mathcal{I}(\text{vs atts rens}, w_i, (W',\mathcal{R}'[\cdot{\rightrightarrows}(w_i, w_{\text{v}})], V'),\mathcal{N}')\\
&\quad \text{where } ((W',\mathcal{R}',V'),\mathcal{N}') = \mathcal{I}(\text{v}, w_v, (W,\mathcal{R},V),\mathcal{N})\\
&\mathcal{I}([\,]\ \text{atts rens}, w_i, (W,\mathcal{R},V),\mathcal{N}) = \mathcal{I}(\text{atts rens}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\mathcal{I}(\text{att} = ((\text{v}_o, \text{port}_o), (\text{v}_t, \text{port}_t))\ \text{atts rens}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \mathcal{I}(\text{atts rens}, w_i, (W \cup \{w_{\text{att}}, w_{\text{PR}_o}, w_{\text{PR}_t}\},\\
&\quad \mathcal{R}[\text{att}{\rightrightarrows}(w_i, w_{\text{att}})][\text{strt}{\rightrightarrows}(w_{\text{att}}, w_{\text{PR}_o})][\text{end}{\rightrightarrows}(w_{\text{att}}, w_{\text{PR}_t})]\\
&\qquad [\text{vref}{\rightrightarrows}\{(w_{\text{PR}_o}, w_{\text{v}_o}), (w_{\text{PR}_t}, w_{\text{v}_t})\}]\\
&\qquad [\text{pref}{\rightrightarrows}\{(w_{\text{PR}_o}, w_{\text{port}_o}), (w_{\text{PR}_t}, w_{\text{port}_t})\}],\\
&\quad V[\text{A}{\rightrightarrows}w_{\text{att}}][\text{PR}{\rightrightarrows}\{w_{\text{PR}_o}, w_{\text{PR}_t}\}]), \mathcal{N})\\
&\quad \text{where } w_{\text{att}} = \mathcal{W}(\text{att}), w_{\text{PR}_o} = \mathcal{W}((\text{v}_o, \text{port}_o)) \text{ and } w_{\text{PR}_t} = \mathcal{W}((\text{v}_t, \text{port}_t))\\
&\mathcal{I}([\,]\ \text{rens}, w_i, (W,\mathcal{R},V),\mathcal{N}) = \mathcal{I}(\text{rens}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\mathcal{I}(\text{ren} = ((\text{v}, \text{port}), \text{id})\ \text{rens}, w_i, (W,\mathcal{R},V),\mathcal{N})\\
&\quad = \mathcal{I}(\text{rens}, w_i, (W \cup \{w_{\text{ren}}, w_{\text{PR}}, w_{\text{id}}\},\\
&\quad \mathcal{R}[\text{ren}{\rightrightarrows}(w_i, w_{\text{ren}})][\text{rend}{\rightrightarrows}(w_{\text{ren}}, w_{\text{PR}})]\\
&\qquad [\text{vref}{\rightrightarrows}(w_{\text{PR}}, w_{\text{v}})][\text{pref}{\rightrightarrows}(w_{\text{PR}}, w_{\text{port}})]\\
&\qquad [\text{prt}{\rightrightarrows}(w_i, w_{\text{ren}})][\text{name}{\rightrightarrows}(w_{\text{ren}}, w_{\text{id}})],\\
&\quad V[\text{R}{\rightrightarrows}w_{\text{ren}}][\text{PR}{\rightrightarrows}w_{\text{PR}}][\text{N}{\rightrightarrows}w_{\text{id}}][\text{id}{\rightrightarrows}w_{\text{ren}}]), \mathcal{N})\\
&\quad \text{where } w_{\text{ren}} = \mathcal{W}(\text{ren}) \text{ and } w_{\text{PR}} = \mathcal{W}((\text{v}, \text{port}))\\
&\mathcal{I}([\,], w_i, (W,\mathcal{R},V),\mathcal{N}) = ((W,\mathcal{R},V),\mathcal{N})^{\square}
\end{aligned}
$$

Appendix B. Translating constraints

The translation of a set of constraints yields a formula map that returns, for a given identifier, the corresponding constraint translated into a modal formula. The function in Definition 11 takes a structural constraint and a formula environment and yields a new formula environment that includes the translation of the constraint. A notational convention adopted to present the translation is to consider terminals and non-terminals of the syntax as sets. For instance, $\text{f} \in F$ is used to indicate that expression f is built according to non-terminal F.

Definition 11 (*Translation: structural constraints*). Let $\mathfrak{f}$ be a formula map with type $FM = ID \to SForm$. Given $\mathfrak{f}$, a structural constraint in *SConst* is translated into a state formula of the graded hybrid logic by function $\mathcal{T}_S\colon SConst \times FM \to FM$ inductively defined as

$$\mathcal{T}_S(\text{fd fds}, \mathfrak{f}) = \mathfrak{f}[\text{id} \mapsto \mathbf{F}_S(\text{f}, \mathcal{T}_S(\text{fds}, \mathfrak{f}))]$$

where $\text{fd} \in FDecl$, $\text{fds} \in FDecl^*$, $\text{id} \in ID$, $\text{f} \in F$, and $\mathbf{F}_S\colon F \times FM \to SForm$ is defined as

$$
\begin{aligned}
&\mathbf{F}_S(\textbf{true}, \mathfrak{f}) = \top\\
&\mathbf{F}_S(\textbf{false}, \mathfrak{f}) = \ \bot\\
&\mathbf{F}_S(\text{p}, \mathfrak{f}) = p\\
&\mathbf{F}_S(\textbf{not}\ \text{f}, \mathfrak{f}) = \neg\mathbf{F}_S(\text{f}, \mathfrak{f})\\
&\mathbf{F}_S(\text{f}\ \textbf{or}\ \text{g}, \mathfrak{f}) = \mathbf{F}_S(\text{f}, \mathfrak{f}) \vee \mathbf{F}_S(\text{g}, \mathfrak{f})\\
&\mathbf{F}_S(\text{f}\ \textbf{and}\ \text{g}, \mathfrak{f}) = \mathbf{F}_S(\text{f}, \mathfrak{f}) \wedge \mathbf{F}_S(\text{g}, \mathfrak{f})\\
&\mathbf{F}_S(\text{f}\ \textbf{implies}\ \text{g}, \mathfrak{f}) = \mathbf{F}_S(\text{f}, \mathfrak{f}) \to \mathbf{F}_S(\text{g}, \mathfrak{f})\\
&\mathbf{F}_S(\text{f}\ \textbf{iff}\ \text{g}, \mathfrak{f}) = \mathbf{F}_S(\text{f}, \mathfrak{f}) \leftrightarrow \mathbf{F}_S(\text{g}, \mathfrak{f})\\
&\mathbf{F}_S([\text{r}]\ \text{f}, \mathfrak{f}) = [\mathbf{R}_S(\text{r})]_0\ \mathbf{F}_S(\text{f}, \mathfrak{f})\\
&\mathbf{F}_S([\text{n}, \text{r}]\text{f}, \mathfrak{f}) = [n, \mathbf{R}_S(\text{r})]_0\ \mathbf{F}_S(\text{f}, \mathfrak{f})\\
&\mathbf{F}_S(\langle\text{r}\rangle\text{f}, \mathfrak{f}) = \langle\mathbf{R}_S(\text{r})\rangle_0\ \mathbf{F}_S(\text{f}, \mathfrak{f})\\
&\mathbf{F}_S(\langle\text{n}, \text{r}\rangle\text{f}, \mathfrak{f}) = \langle n, \mathbf{R}_S(\text{r})\rangle_0\ \mathbf{F}_S(\text{f}, \mathfrak{f})\\
&\mathbf{F}_S(\text{id}\ \text{a}_1 \ldots \text{a}_n, \mathfrak{f}) = \mathfrak{f}(\text{id})[\text{f}_1 \ldots \text{f}_n / \text{a}_1 \ldots \text{a}_n]
\end{aligned}
$$

$$\mathbf{F}_S(\mathrm{i}, \mathfrak{f}) = i$$
$$\mathbf{F}_S(\textbf{at } \mathrm{i}, \mathfrak{f}) = @_i$$
$$\mathbf{F}_S(\mathrm{cvar}, \mathfrak{f}) = cvar$$
$$\mathbf{F}_S(\textbf{at } \mathrm{cvar}, \mathfrak{f}) = @_{cvar}$$
$$\mathbf{F}_S(\textbf{exists } \mathrm{cvar{:}type.f}, \mathfrak{f}) = \bigvee_{i \in Nom_{type}} @_i \, \mathbf{F}_S(\mathrm{f}, \mathfrak{f})[cvar/i]$$
$$\mathbf{F}_S(\textbf{exists } \mathrm{cvar.f}, \mathfrak{f}) = \bigvee_{i \in Nom} @_i \, \mathbf{F}_S(\mathrm{f}, \mathfrak{f})[cvar/i]$$
$$\mathbf{F}_S(\textbf{forall } \mathrm{cvar{:}type.f}, \mathfrak{f}) = \bigwedge_{i \in Nom_{type}} @_i \, \mathbf{F}_S(\mathrm{f}, \mathfrak{f})[cvar/i]$$
$$\mathbf{F}_S(\textbf{forall } \mathrm{cvar.f}, \mathfrak{f}) = \bigwedge_{i \in Nom} @_i \, \mathbf{F}_S(\mathrm{f}, \mathfrak{f})[cvar/i]$$

where p is a proposition, $\mathrm{g} \in F$, $\mathrm{r} \in RelF$, $\mathrm{n} \in \mathbb{N}$, $\mathrm{a}_1 \ldots \mathrm{a}_n$ are actual parameters, $\mathrm{f}_1 \ldots \mathrm{f}_n$ are formal parameters, $\mathrm{i} \in Nom$, cvar is a variable, type is a type defined by a pattern, and with function $\mathbf{R}_S : RelF \to RForm$ inductively defined as

$$\mathbf{R}_S(\mathrm{s}) = \mathbf{M}_S(\mathrm{s}), \quad \mathbf{R}_S(\mathrm{r}_1.\mathrm{r}_2) = \mathbf{R}_S(\mathrm{r}_1).\mathbf{R}_S(\mathrm{r}_2)$$
$$\mathbf{R}_S(\mathrm{r}_1 + \mathrm{r}_2) = \mathbf{R}_S(\mathrm{r}_1) + \mathbf{R}_S(\mathrm{r}_2), \quad \mathbf{R}_S(\mathrm{r}^*) = \mathbf{R}_S(\mathrm{r})^*$$
$$\mathbf{R}_S(\mathrm{r}^+) = \mathbf{R}_S(\mathrm{r})^+$$

with $\mathrm{s} \in SymF$, $\mathrm{r}_1, \mathrm{r}_2 \in RelF$, and $\mathbf{M}_S : SymF \to MForm$ is defined as

$$\mathbf{M}_S(\mathrm{m}) = m, \quad \mathbf{M}_S(\textbf{not } \mathrm{s}) = \neg \, \mathbf{M}_S(\mathrm{s})$$
$$\mathbf{M}_S(\textbf{true}) = \top, \quad \mathbf{M}_S(\textbf{false}) = \bot$$
$$\mathbf{M}_S(\mathrm{s}_1 \textbf{ and } \mathrm{s}_2) = \mathbf{M}_S(\mathrm{s}_1) \wedge \mathbf{M}_S(\mathrm{s}_2), \quad \mathbf{M}_S(\mathrm{s}_1 \textbf{ or } \mathrm{s}_2) = \mathbf{M}_S(\mathrm{s}_1) \vee \mathbf{M}_S(\mathrm{s}_2)$$

for m an atomic modal symbol, and $\mathrm{s}_1, \mathrm{s}_2 \in SymF$.□

Definition 12 (*Translation: reconfiguration constraints*). Given a formula map $\mathfrak{f}$, a reconfiguration constraint in *RConst* is translated into a state formula of the 2-layer graded hybrid logic by functions

$$\mathbf{T}_R : RConst \times FM \to FM, \quad \mathbf{F}_R : F \times FM \to SForm,$$
$$\mathbf{R}_R : RelF \to RForm, \text{ and } \mathbf{M}_R : SymF \to MForm.$$

The clauses of $\mathbf{T}_R$ coincide with the clauses of $\mathbf{T}_S$. The clauses of $\mathbf{F}_R$ coincide with the clauses of $\mathbf{F}_S$ with the exceptions as follows: clauses that translate hybrid features, which are elided; clauses that translate quantifiers, which are replaced by clauses

$$\mathbf{F}_R(\textbf{exists } \mathrm{cvar{:}type.f}, \mathfrak{f}) = \exists \; cvar{:}type.\mathbf{F}_R(\mathrm{f}, \mathfrak{f})$$
$$\mathbf{F}_R(\textbf{exists } \mathrm{cvar.f}, \mathfrak{f}) = \exists \; cvar.\mathbf{F}_R(\mathrm{f}, \mathfrak{f})$$
$$\mathbf{F}_R(\textbf{forall } \mathrm{cvar{:}type.f}, \mathfrak{f}) = \forall \; cvar{:}type.\mathbf{F}_R(\mathrm{f}, \mathfrak{f})$$
$$\mathbf{F}_R(\textbf{forall } \mathrm{cvar.f}, \mathfrak{f}) = \forall \; cvar.\mathbf{F}_R(\mathrm{f}, \mathfrak{f});$$

and an added clause for translating nested constraints

$$\mathbf{F}_R(\textbf{nest } \mathrm{id}, \mathfrak{f}) = \mathfrak{f}(\mathrm{id}).$$

Function $\mathbf{R}_R$ coincides with function $\mathbf{R}_S$. The clauses of function $\mathbf{M}_R$ coincide with the clauses of $\mathbf{M}_S$ with the addition of a clause for translating structured symbols in Fig. 10 matching reconfiguration operations as follows:

$$\mathbf{M}^R(\mathrm{m}[\mathrm{a}_1 \ldots \mathrm{a}_n], \mathfrak{f}) = m(a_1 \ldots a_n).$$

where $\mathrm{m} \in$ ASYM and $\mathrm{a}_j \in$ Filter.□

References

[1] Medvidovic N. Adls and dynamic architecture changes. In: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops. New York, NY, USA: ACM Press; 1996. p. 24–7.

[2] Medvidovic N, Taylor R. A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 2000;26(1):70–93.

[3] Sanchez A, Barbosa LS, Riesco D. A language for behavioural modelling of architectural patterns. In: Proceedings of the third workshop on behavioural modelling, BM-FA '11. New York, NY, USA: ACM; 2011. p. 17–24.

[4] Sanchez A, Barbosa LS, Riesco D. Bigraphical modelling of architectural patterns, In: The eighth international symposium on formal aspects of component software, FACS 2011, Oslo, Norway, September 14–16, 2011, Revised selected papers, Lecture notes in computer science, vol. 7253. Berlin, Heidelberg: Springer; 2012. p. 313–30.

[5] van Deursen A, Klint P, Visser J. Domain-specific languages: an annotated bibliography. SIGPLAN Not. 2000;35(6):26–36.

[6] Sanchez A, Barbosa LS, Riesco D. Verifying bigraphical models of architectural reconfigurations. In: The seventh international symposium on theoretical aspects of software engineering, TASE 2013, 1–3 July 2013, Birmingham, UK. USA: IEEE Computer Society; 2013. p. 135–8.

[7] Sanchez A, Barbosa LS, Riesco D. Specifying structural constraints of architectural patterns in the archery language. In: Simos TE, Tsitouras C, editors. Proceedings of the international conference on numerical analysis and applied mathematics 2014 (ICNAAM-2014), vol. 1648, AIP proceedings; 2015. p. 310008(1)–(5).
[8] Sanchez A, Barbosa LS, Madeira A. Modelling and verifying smell-free architectures with the archery language. In: Canal C, Idani A, editors. Software engineering and formal methods, Lecture notes in computer science, vol. 8938. Switzerland: Springer International Publishing; 2015. p. 147–63.
[9] Madeira A, Martins MA, Barbosa LS, Hennicker R. Refinement in hybridised institutions. Form Asp Comput 2015;27(2):375–95.
[10] Sanchez A. A calculus of architectural patterns (to appear) [Ph.D. thesis], Universidad Nacional de San Luis; 2015.
[11] Services UKB. Handbook of transfusion medicine. 5th edition. United Kingdom: The Stationery Office; 2013.
[12] Mellor SJ, Kendall S, Uhl A, Weise D. MDA distilled. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.; 2004.
[13] van der Hoek W. On the semantics of graded modalities. J Appl Non-Class Logics 1992;2(1).
[14] Areces C, Blackburn P, Marx M. Hybrid logics: characterization, interpolation and complexity. J Symb Log 2001;66(3):977–1010.
[15] Divroodi AR, Nguyen LA. On bisimulations for description logics. Inf Sci 2015;295:465–93.
[16] Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. Pattern-oriented software architecture volume 1: a system of patterns, 1st edition. England: Wiley; 1996.
[17] Shaw M, Garlan D. Software architecture: perspectives on an emerging discipline.USA: Prentice Hall; 1996.
[18] Garlan D, Monroe R, Wile D. ACME: an architecture description interchange language. In: Proceedings of the 1997 conference of the centre for advanced studies on collaborative research, CASCON '97. IBM Press; 1997. p. 169–83.
[19] Magee J, Kramer J. Dynamic structure in software architectures. In: Proceedings of the fourth ACM SIGSOFT symposium on foundations of software engineering, SIGSOFT '96. New York, NY, USA: ACM; 1996. p. 3–14.
[20] Georgiadis I, Magee J, Kramer J. Self-organising software architectures for distributed systems. In: Proceedings of the first workshop on self-healing systems, WOSS '02. New York, NY, USA: ACM; 2002. p. 33–8.
[21] Kramer J, Magee J, Uchitel S. Software architecture modeling and analysis: a rigorous approach. In: Bernardo M, Inverardi P, editors. Formal methods for software architectures, Lecture notes in computer science, vol. 2804. Berlin, Heidelberg: Springer; 2003. p. 44–51.
[22] Allen R, Garlan D. A formal basis for architectural connection. ACM Trans Softw Eng Methodol 1997;6(3):213–49.
[23] Allen R, Douence R, Garlan D. Specifying and analyzing dynamic software architectures. In: Astesiano E, editor. Fundamental approaches to software engineering, Lecture notes in computer science, vol. 1382. Berlin, Heidelberg: Springer; 1998. p. 21–37.
[24] Bruni R, Lluch Lafuente A, Montanari U, Tuosto E. Style based architectural reconfigurations. Bull Eur Assoc Theor Comput Sci 2008;94:161–80.
[25] Bradbury JS, Cordy JR, Dingel J, Wermelinger M. A survey of self-management in dynamic software architecture specifications. In: Proceedings of the first ACM SIGSOFT workshop on self-managed systems, WOSS '04. New York, NY, USA: ACM; 2004. p. 28–33.
[26] Groote JF, Mathijssen A, Reniers M, Usenko Y, van Weerdenburg M. The formal specification language . In: Methods for modelling software systems: Dagstuhl seminar 06351; 2007.
[27] Milner R. Bigraphical reactive systems. In: Larsen KG, Nielsen M, editors. CONCUR, Lecture notes in computer science, vol. 2154. Berlin, Heidelberg, Germany: Springer; 2001. p. 16–35.
[28] Medvidovic N, Rosenblum DS, Redmiles DF, Robbins JE. Modeling software architectures in the unified modeling language. ACM Trans Softw Eng Methodol 2002;11(1):2–57.
[29] Le Métayer D. Describing software architecture styles using graph grammars. IEEE Trans Softw Eng 1998;24:521–33.
[30] Kim JS, Garlan D. Analyzing architectural styles with alloy. In: Proceedings of the ISSTA 2006 workshop on role of software architecture for testing and analysis, ROSATEA '06. New York, NY, USA: ACM; 2006. p. 70–80.
[31] Jackson D. Alloy: a lightweight object modelling notation. ACM Trans Softw Eng Methodol 2002;11(2):256–90.
[32] Dormoy J, Kouchnarenko O, Lanoix A. Using temporal logic for dynamic reconfigurations of components. In: Barbosa LS, Lumpe M, editors. Formal aspects of component software—the seventh international workshop, FACS 2010, Guimarães, Portugal, October 14–16, 2010, Revised selected papers, Lecture notes in computer science, vol. 6921. Berlin, Heidelberg, Germany: Springer; 2010. p. 200–17.
[33] Aguirre N, Maibaum TSE. A temporal logic approach to the specification of reconfigurable component-based systems. In: The 17th IEEE international conference on automated software engineering (ASE 2002), 23–27 September 2002, Edinburgh, Scotland, UK; 2002. p. 271–4.
[34] Canal C, Cámara J, Salaün G. Structural reconfiguration of systems under behavioral adaptation. Sci Comput Program 2012;78(1):46–64.